Unlockd Finance - NFTBatchTransfer

Smart Contract Security Assessment

Prepared by: **Halborn**

Date of Engagement: **September 15th, 2023 - September 19th, 2023**

Visit: **Halborn.com**

DOCUMENT REVISION HISTORY 3
CONTACTS 3
1 EXECUTIVE OVERVIEW 4
1.1 INTRODUCTION 5
1.2 ASSESSMENT SUMMARY 5
1.3 TEST APPROACH & METHODOLOGY 5
2 RISK METHODOLOGY 7
2.1 EXPLOITABILITY 8
2.2 IMPACT 9
2.3 SEVERITY COEFFICIENT 11
2.4 SCOPE 13
3 ASSESSMENT SUMMARY & FINDINGS OVERVIEW 14
4 FINDINGS & TECH DETAILS 15
4.1 (HAL-01) CRYPTOPUNKS CAN BE STOLEN THROUGH THE BATCHPUNKTRANSFERFROM FUNCTION - CRITICAL(10) 17
Description 17
BVSS 19
Recommendation 19
Remediation Plan 19
4.2 (HAL-02) UNNECESSARY GAS CONSUMPTION CHECKS - INFORMATIONAL(0.0) 21
Description 21
BVSS 24
Recommendation 24
Remediation Plan 24

5	RECOMMENDATIONS OVERVIEW	25
6	AUTOMATED TESTING	27
6.1	STATIC ANALYSIS REPORT	28
	Description	28
	Slither results	28
6.2	AUTOMATED SECURITY SCAN	29
	Description	29
	MythX results	29

DOCUMENT REVISION HISTORY

VERSION	MODIFICATION	DATE	AUTHOR
0.1	Document Creation	09/15/2023	Roberto Reigada
0.2	Document Updates	09/18/2023	Roberto Reigada
1.0	Remediation Plan	09/18/2023	Roberto Reigada
1.1	Remediation Plan Review	09/19/2023	Gokberk Gulgun
1.1	Remediation Plan Review	09/19/2023	Gabi Urrutia

CONTACTS

CONTACT	COMPANY	EMAIL
Rob Behnke	Halborn	Rob.Behnke@halborn.com
Steven Walbroehl	Halborn	Steven.Walbroehl@halborn.com
Gabi Urrutia	Halborn	Gabi.Urrutia@halborn.com
Gokberk Gulgun	Halborn	Gokberk.Gulgun@halborn.com
Roberto Reigada	Halborn	Roberto.Reigada@halborn.com

EXECUTIVE OVERVIEW

1.1 INTRODUCTION

Unlockd Finance engaged Halborn to conduct a security assessment on their smart contracts beginning on September 15th, 2023 and ending on September 19th, 2023. The security assessment was scoped to the smart contracts provided in the following GitHub repository:

- UnlockdFinance/NFTBatchTransfers

1.2 ASSESSMENT SUMMARY

The team at Halborn was provided about a week for the engagement and assigned a full-time security engineer to verify the security of the smart contracts. The security engineer is a blockchain and smart-contract security expert with advanced penetration testing, smart-contract hacking and deep knowledge of multiple blockchain protocols.

The purpose of this assessment is to:

- Ensure that smart contract functions operate as intended.
- Identify potential security issues with the smart contracts.

In summary, Halborn identified some security risks that were successfully addressed by the Unlockd Finance team.

1.3 TEST APPROACH & METHODOLOGY

Halborn performed a combination of manual and automated security testing to balance efficiency, timeliness, practicality, and accuracy in regard to the scope of this assessment. While manual testing is recommended to uncover flaws in logic, process, and implementation; automated testing techniques help enhance coverage of the code and can quickly identify items that do not follow the security best practices. The following phases and associated tools were used during the assessment:

- Research into architecture and purpose.
- Smart contract manual code review and walkthrough.
- Graphing out functionality and contract logic/connectivity/functions. (solgraph)
- Manual assessment of use and safety for the critical Solidity variables and functions in scope to identify any arithmetic related vulnerability classes.
- Manual testing by custom scripts.
- Scanning of solidity files for vulnerabilities, security hot-spots or bugs. (MythX)
- Static Analysis of security for scoped contract, and imported functions. (Slither)
- Testnet deployment. (Foundry)

2. RISK METHODOLOGY

Every vulnerability and issue observed by Halborn is ranked based on **two sets** of **Metrics** and a **Severity Coefficient**. This system is inspired by the industry standard Common Vulnerability Scoring System.

The two **Metric sets** are: **Exploitability** and **Impact**. **Exploitability** captures the ease and technical means by which vulnerabilities can be exploited and **Impact** describes the consequences of a successful exploit.

The **Severity Coefficients** is designed to further refine the accuracy of the ranking with two factors: **Reversibility** and **Scope**. These capture the impact of the vulnerability on the environment as well as the number of users and smart contracts affected.

The final score is a value between 0-10 rounded up to 1 decimal place and 10 corresponding to the highest security risk. This provides an objective and accurate rating of the severity of security vulnerabilities in smart contracts.

The system is designed to assist in identifying and prioritizing vulnerabilities based on their level of risk to address the most critical issues in a timely manner.

2.1 EXPLOITABILITY

Attack Origin (AO):

Captures whether the attack requires compromising a specific account.

Attack Cost (AC):

Captures the cost of exploiting the vulnerability incurred by the attacker relative to sending a single transaction on the relevant blockchain. Includes but is not limited to financial and computational cost.

Attack Complexity (AX):

Describes the conditions beyond the attacker's control that must exist in order to exploit the vulnerability. Includes but is not limited to macro situation, available third-party liquidity and regulatory challenges.

Metrics:

Exploitability Metric (m_E)	Metric Value	Numerical Value
Attack Origin (AO)	Arbitrary (AO:A)	1
	Specific (AO:S)	0.2
Attack Cost (AC)	Low (AC:L)	1
	Medium (AC:M)	0.67
	High (AC:H)	0.33
Attack Complexity (AX)	Low (AX:L)	1
	Medium (AX:M)	0.67
	High (AX:H)	0.33

Exploitability E is calculated using the following formula:

$$E = \prod m_e$$

EXECUTIVE OVERVIEW

2.2 IMPACT

Confidentiality (C):

Measures the impact to the confidentiality of the information resources managed by the contract due to a successfully exploited vulnerability. Confidentiality refers to limiting access to authorized users only.

Integrity (I):

Measures the impact to integrity of a successfully exploited vulnerability. Integrity refers to the trustworthiness and veracity of data stored and/or processed on-chain. Integrity impact directly affecting Deposit or Yield records is excluded.

Availability (A):

Measures the impact to the availability of the impacted component resulting from a successfully exploited vulnerability. This metric refers to smart contract features and functionality, not state. Availability impact directly affecting Deposit or Yield is excluded.

Deposit (D):

Measures the impact to the deposits made to the contract by either users or owners.

Yield (Y):

Measures the impact to the yield generated by the contract for either users or owners.

Metrics:

Impact Metric (m_I)	Metric Value	Numerical Value
Confidentiality (C)	None (I:N)	0
	Low (I:L)	0.25
	Medium (I:M)	0.5
	High (I:H)	0.75
	Critical (I:C)	1
Integrity (I)	None (I:N)	0
	Low (I:L)	0.25
	Medium (I:M)	0.5
	High (I:H)	0.75
	Critical (I:C)	1
Availability (A)	None (A:N)	0
	Low (A:L)	0.25
	Medium (A:M)	0.5
	High (A:H)	0.75
	Critical	1
Deposit (D)	None (D:N)	0
	Low (D:L)	0.25
	Medium (D:M)	0.5
	High (D:H)	0.75
	Critical (D:C)	1
Yield (Y)	None (Y:N)	0
	Low (Y:L)	0.25
	Medium: (Y:M)	0.5
	High: (Y:H)	0.75
	Critical (Y:H)	1

Impact I is calculated using the following formula:

$$I = max(m_I) + \frac{\sum m_I - max(m_I)}{4}$$

2.3 SEVERITY COEFFICIENT

Reversibility (R):

Describes the share of the exploited vulnerability effects that can be reversed. For upgradeable contracts, assume the contract private key is available.

Scope (S):

Captures whether a vulnerability in one vulnerable contract impacts resources in other contracts.

Coefficient (C)	Coefficient Value	Numerical Value
Reversibility (r)	None (R:N) Partial (R:P) Full (R:F)	1 0.5 0.25
Scope (s)	Changed (S:C) Unchanged (S:U)	1.25 1

Severity Coefficient C is obtained by the following product:

$$C = rs$$

The Vulnerability Severity Score S is obtained by:

$$S = min(10, EIC * 10)$$

The score is rounded up to 1 decimal places.

Severity	Score Value Range
Critical	9 - 10
High	7 - 8.9
Medium	4.5 - 6.9
Low	2 - 4.4
Informational	0 - 1.9

2.4 SCOPE

1. IN-SCOPE TREE & COMMIT :

The security assessment was scoped to the following smart contracts:

GitHub repository: UnlockdFinance/NFTBatchTransfers
Commit ID: 55849e99794b9f12bedd561bbe0c3ba7f8176549
Fixed Commit ID: c885c4fedf5e8840dc25999594cc5b0f6f7ee677.
Smart contracts in scope:

- NFTBatchTransfer.sol

3. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
1	0	0	0	1

EXECUTIVE OVERVIEW

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
(HAL-01) CRYPTOPUNKS CAN BE STOLEN THROUGH THE BATCHPUNKTRANSFERFROM FUNCTION	Critical (10)	SOLVED - 09/18/2023
(HAL-02) UNNECESSARY GAS CONSUMPTION CHECKS	Informational (0.0)	SOLVED - 09/18/2023

FINDINGS & TECH DETAILS

4.1 (HAL-01) CRYPTOPUNKS CAN BE STOLEN THROUGH THE BATCHPUNKTRANSFERFROM FUNCTION - CRITICAL(10)

Commit IDs affected:
- 55849e99794b9f12bedd561bbe0c3ba7f8176549

Description:

The contract NFTBatchTransfer implements the function batchPunkTransferFrom() that allows transferring CryptoPunks in batches:

Listing 1: NFTBatchTransfer.sol (Lines 102-114)

```
/**
 * @dev Manages a batch transfer of NFTs, specifically tailored for CryptoPunks alongside other standard ERC721 NFTs.
 * @param nftTransfers An array of NftTransfer structs specifying the NFTs for transfer.
 * @param to The destination address for the NFTs.
 */
function batchPunkTransferFrom(
    NftTransfer[] calldata nftTransfers,
    address to
) external payable {
    uint256 length = nftTransfers.length;
    uint256 gasLeftStart = gasleft();
    bool success;

    // Process batch transfers, differentiate between CryptoPunks and standard ERC721 tokens.
    for (uint i = 0; i < length;) {
        address contractAddr = nftTransfers[i].contractAddress;
        uint256 tokenId = nftTransfers[i].tokenId;

        if (contractAddr != punkContract) {
            // If it's not a CryptoPunk, use the standard ERC721 `transferFrom` function.
```

```
            (success, ) = contractAddr.call(
                abi.encodeWithSignature(
                    "transferFrom(address,address,uint256)",
                    msg.sender,
                    to,
                    tokenId
                )
            );
        } else {
            // If it's a CryptoPunk, first the contract buy the punk to be allowed to transfer it.
            (success, ) = punkContract.call{value: 0}(
                abi.encodeWithSignature("buyPunk(uint256)", tokenId)
            );

            // Once the punk is owned by the contract, the transfer method is executed
            (success, ) = punkContract.call(
                abi.encodeWithSignature(
                    "transferPunk(address,uint256)",
                    to,
                    tokenId
                )
            );
        }

        unchecked {
            i++;
        }

        // Check the transfer status and gas consumption.
        if (!success || gasleft() < gasLeftStart / 2) {
            revert("Transfer failed");
        }

    }
}
```

As we can see in the code above, the contract first buys the CryptoPunk to then transfers it to the `to` address. This is needed as the CryptoPunks contract lacks the standard `ERC721.approve()` and `ERC721.transferFrom()` functions. Although, with the current implementation, the following attack vector would be possible:

1. Alice wants to transfer her CryptoPunk #1337 through the NFTBatchTransfer.batchPunkTransferFrom() function.
2. Alice calls contract_CryptoPunks.offerPunkForSaleToAddress(1337, 0, <NftBatchTransfer address>) to create a sale with no cost that can only be bought by the NFTBatchTransfer contract.
3. Alice calls contract_NFTBatchTransfer.batchPunkTransferFrom([(<CryptoPunk address>, 1337)], <Bob address>) to transfer her CryptoPunk #1337 to Bob.
4. The attacker frontruns the transaction and calls contract_NFTBatchTransfer.batchPunkTransferFrom([(<CryptoPunk address>, 1337)], <Bob address>) stealing the CryptoPunk #1337.

```
ALICE calls: << contract_PUNK.offerPunkForSaleToAddress(2859, 0, address(contract_NFTBatchTransfer)) >>
  contract_PUNK.balanceOf(ALICE) -> 144
  contract_PUNK.balanceOf(ATTACKER) -> 0

ATTACKER calls: << contract_NFTBatchTransfer.batchPunkTransferFrom(_nftTransfers, attacker) >>
  contract_PUNK.balanceOf(ALICE) -> 143
  contract_PUNK.balanceOf(ATTACKER) -> 1
```

BVSS:

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:C/Y:N/R:N/S:C (10)

Recommendation:

It is recommended to update the batchPunkTransferFrom() function, so it checks the ownership of each CryptoPunk ID transferred through the usage of the punkIndexToAddress public mapping. msg.sender should always be the owner of the CryptoPunk ID being transferred through the batchPunkTransferFrom() function.

Remediation Plan:

SOLVED: The Unlockd Finance team solved the issue by implementing the recommended solution.

Commit ID : c885c4fedf5e8840dc25999594cc5b0f6f7ee677.

4.2 (HAL-02) UNNECESSARY GAS CONSUMPTION CHECKS - INFORMATIONAL (0.0)

Commit IDs affected:
- 55849e99794b9f12bedd561bbe0c3ba7f8176549

Description:

The contract NFTBatchTransfer implements some gas consumption checks in the batchTransferFrom() and batchPunkTransferFrom() functions:

Listing 2: NFTBatchTransfer.sol (Line 62)

```
/**
 * @dev Orchestrates a batch transfer of standard ERC721 NFTs.
 * @param nftTransfers An array of NftTransfer structs detailing the NFTs to be moved.
 * @param to The recipient's address.
 */
function batchTransferFrom(
    NftTransfer[] calldata nftTransfers,
    address to
) external payable {
    uint256 length = nftTransfers.length;

    // Capturing the initial gas at the start for later comparisons.
    uint256 gasLeftStart = gasleft();

    // Iterate through each NFT in the array to facilitate the transfer.
    for (uint i = 0; i < length;) {
        address contractAddress = nftTransfers[i].contractAddress;
        uint256 tokenId = nftTransfers[i].tokenId;

        // Dynamically call the `transferFrom` function on the target ERC721 contract.
        (bool success, ) = contractAddress.call(
            abi.encodeWithSignature(
```

```
                "transferFrom(address,address,uint256)",
                msg.sender,
                to,
                tokenId
            )
        );

        // Check the transfer status and gas consumption.
        if (!success || gasleft() < gasLeftStart / 2) {
            revert("Gas too low");
        }

        // Use unchecked block to bypass overflow checks for efficiency.
        unchecked {
            i++;
        }
    }
}
```

Listing 3: NFTBatchTransfer.sol (Line 122)

```
/**
 * @dev Manages a batch transfer of NFTs, specifically tailored for CryptoPunks alongside other standard ERC721 NFTs.
 * @param nftTransfers An array of NftTransfer structs specifying the NFTs for transfer.
 * @param to The destination address for the NFTs.
 */
function batchPunkTransferFrom(
    NftTransfer[] calldata nftTransfers,
    address to
) external payable {
    uint256 length = nftTransfers.length;
    uint256 gasLeftStart = gasleft();
    bool success;

    // Process batch transfers, differentiate between CryptoPunks and standard ERC721 tokens.
    for (uint i = 0; i < length;) {
        address contractAddr = nftTransfers[i].contractAddress;
        uint256 tokenId = nftTransfers[i].tokenId;

        if (contractAddr != punkContract) {
```

```
            // If it's not a CryptoPunk, use the standard ERC721 `transferFrom` function.
            (success, ) = contractAddr.call(
                abi.encodeWithSignature(
                    "transferFrom(address,address,uint256)",
                    msg.sender,
                    to,
                    tokenId
                )
            );
        } else {
            // If it's a CryptoPunk, first the contract buy the punk to be allowed to transfer it.
            (success, ) = punkContract.call{value: 0}(
                abi.encodeWithSignature("buyPunk(uint256)", tokenId)
            );

            // Once the punk is owned by the contract, the transfer method is executed
            (success, ) = punkContract.call(
                abi.encodeWithSignature(
                    "transferPunk(address,uint256)",
                    to,
                    tokenId
                )
            );
        }

        unchecked {
            i++;
        }

        // Check the transfer status and gas consumption.
        if (!success || gasleft() < gasLeftStart / 2) {
            revert("Transfer failed");
        }

    }
}
```

These checks are redundant and can be removed. The users' wallet will simulate the transaction and determine what should be the maximum gas limit for the transaction.

BVSS:

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:N/S:C (0.0)

Recommendation:

It is recommended to remove the gas consumption checks. Instead, a max. amount of NFTs per batch transfer can be enforced at the smart contract level.

Remediation Plan:

SOLVED: The Unlockd Finance team solved the issue by removing all the gas consumption checks.

Commit ID : c885c4fedf5e8840dc25999594cc5b0f6f7ee677.

RECOMMENDATIONS OVERVIEW

1. Update the batchPunkTransferFrom() function, so it checks the ownership of each CryptoPunk ID transferred through the use of the punkIndexToAddress public mapping. msg.sender should always be the owner of the CryptoPunk ID being transferred through the batchPunkTransferFrom() function.
2. Remove the gas consumption checks. Instead, a max. amount of NFTs per batch transfer can be enforced at the smart contract level.

AUTOMATED TESTING

6.1 STATIC ANALYSIS REPORT

Description:

Halborn used automated testing techniques to enhance the coverage of certain areas of the smart contracts in scope. Among the tools used was Slither, a Solidity static analysis framework. After Halborn verified the smart contracts in the repository and was able to compile them correctly into their ABIS and binary format, Slither was run against the contracts. This tool can statically verify mathematical relationships between Solidity variables to detect invalid or inconsistent usage of the contracts' APIs across the entire code-base.

Slither results:

NFTBatchTransfer.sol

```
INFO:Detectors:
NFTBatchTransfer.batchPunkTransferFrom(NFTBatchTransfer.NftTransfer[],address) (src/NFTBatchTransfer.sol#78-127) sends eth to arbitrary user
        Dangerous calls:
        - (success,None) = punkContract.call{value: 0}(abi.encodeWithSignature(buyPunk(uint256),tokenId)) (src/NFTBatchTransfer.sol#103-105)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#functions-that-send-ether-to-arbitrary-destinations
INFO:Detectors:
Contract locking ether found:
        Contract NFTBatchTransfer (src/NFTBatchTransfer.sol#13-138) has payable functions:
         - NFTBatchTransfer.batchTransferFrom(NFTBatchTransfer.NftTransfer[],address) (src/NFTBatchTransfer.sol#37-71)
         - NFTBatchTransfer.batchPunkTransferFrom(NFTBatchTransfer.NftTransfer[],address) (src/NFTBatchTransfer.sol#78-127)
         - NFTBatchTransfer.fallback() (src/NFTBatchTransfer.sol#130-132)
         - NFTBatchTransfer.receive() (src/NFTBatchTransfer.sol#135-137)
        But does not have a function to withdraw the ether
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#contracts-that-lock-ether
INFO:Detectors:
NFTBatchTransfer.batchPunkTransferFrom(NFTBatchTransfer.NftTransfer[],address).success (src/NFTBatchTransfer.sol#84) is written in both
        (success,None) = punkContract.call{value: 0}(abi.encodeWithSignature(buyPunk(uint256),tokenId)) (src/NFTBatchTransfer.sol#103-105)
        (success,None) = punkContract.call(abi.encodeWithSignature(transferPunk(address,uint256),to,tokenId)) (src/NFTBatchTransfer.sol#108-114)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#write-after-write
INFO:Detectors:
Function NFTBatchTransfer.batchTransferFrom(NFTBatchTransfer.NftTransfer[],address) (src/NFTBatchTransfer.sol#37-71) contains a low level call to a custom address
Function NFTBatchTransfer.batchPunkTransferFrom(NFTBatchTransfer.NftTransfer[],address) (src/NFTBatchTransfer.sol#78-127) contains a low level call to a custom address
Reference: https://github.com/pessimistic-io/slitherin/blob/master/docs/call_forward_to_protected.md
INFO:Detectors:
NFTBatchTransfer.constructor(address)._punkContract (src/NFTBatchTransfer.sol#28) lacks a zero-check on :
                - punkContract = _punkContract (src/NFTBatchTransfer.sol#29)
NFTBatchTransfer.batchTransferFrom(NFTBatchTransfer.NftTransfer[],address).to (src/NFTBatchTransfer.sol#39) lacks a zero-check on :
                - (success) = contractAddress.call(abi.encodeWithSignature(transferFrom(address,address,uint256),msg.sender,to,tokenId)) (src/NFTBatchTransfer.sol#52-59)
NFTBatchTransfer.batchTransferFrom(NFTBatchTransfer.NftTransfer[],address).contractAddress (src/NFTBatchTransfer.sol#48) lacks a zero-check on :
                - (success) = contractAddress.call(abi.encodeWithSignature(transferFrom(address,address,uint256),msg.sender,to,tokenId)) (src/NFTBatchTransfer.sol#52-59)
NFTBatchTransfer.batchPunkTransferFrom(NFTBatchTransfer.NftTransfer[],address).to (src/NFTBatchTransfer.sol#80) lacks a zero-check on :
                - (success,None) = contractAddr.call(abi.encodeWithSignature(transferFrom(address,address,uint256),msg.sender,to,tokenId)) (src/NFTBatchTransfer.sol#93-100)
                - (success,None) = punkContract.call(abi.encodeWithSignature(transferPunk(address,uint256),to,tokenId)) (src/NFTBatchTransfer.sol#108-114)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#missing-zero-address-validation
INFO:Detectors:
NFTBatchTransfer.batchTransferFrom(NFTBatchTransfer.NftTransfer[],address) (src/NFTBatchTransfer.sol#37-71) has external calls inside a loop: (success) = contractAddress.call(abi.encodeWithSignature(transferFrom(address,address,uint256),
msg.sender,to,tokenId)) (src/NFTBatchTransfer.sol#52-59)
NFTBatchTransfer.batchPunkTransferFrom(NFTBatchTransfer.NftTransfer[],address) (src/NFTBatchTransfer.sol#78-127) has external calls inside a loop: (success,None) = contractAddr.call(abi.encodeWithSignature(transferFrom(address,address,ui
nt256),msg.sender,to,tokenId)) (src/NFTBatchTransfer.sol#93-100)
NFTBatchTransfer.batchPunkTransferFrom(NFTBatchTransfer.NftTransfer[],address) (src/NFTBatchTransfer.sol#78-127) has external calls inside a loop: (success,None) = punkContract.call{value: 0}(abi.encodeWithSignature(buyPunk(uint256),toke
nId)) (src/NFTBatchTransfer.sol#103-105)
NFTBatchTransfer.batchPunkTransferFrom(NFTBatchTransfer.NftTransfer[],address) (src/NFTBatchTransfer.sol#78-127) has external calls inside a loop: (success,None) = punkContract.call(abi.encodeWithSignature(transferPunk(address,uint256),t
o,tokenId)) (src/NFTBatchTransfer.sol#108-114)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation/#calls-inside-a-loop
INFO:Detectors:
Pragma version0.8.19 (src/NFTBatchTransfer.sol#2) necessitates a version too recent to be trusted. Consider deploying with 0.8.18.
solc-0.8.19 is not recommended for deployment
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-versions-of-solidity
INFO:Detectors:
Low level call in NFTBatchTransfer.batchTransferFrom(NFTBatchTransfer.NftTransfer[],address) (src/NFTBatchTransfer.sol#37-71):
        - (success) = contractAddress.call(abi.encodeWithSignature(transferFrom(address,address,uint256),msg.sender,to,tokenId)) (src/NFTBatchTransfer.sol#52-59)
Low level call in NFTBatchTransfer.batchPunkTransferFrom(NFTBatchTransfer.NftTransfer[],address) (src/NFTBatchTransfer.sol#78-127):
        - (success,None) = contractAddr.call(abi.encodeWithSignature(transferFrom(address,address,uint256),msg.sender,to,tokenId)) (src/NFTBatchTransfer.sol#93-100)
        - (success,None) = punkContract.call{value: 0}(abi.encodeWithSignature(buyPunk(uint256),tokenId)) (src/NFTBatchTransfer.sol#103-105)
        - (success,None) = punkContract.call(abi.encodeWithSignature(transferPunk(address,uint256),to,tokenId)) (src/NFTBatchTransfer.sol#108-114)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#low-level-calls

INFO:Detectors:
In a function NFTBatchTransfer.batchPunkTransferFrom(NFTBatchTransfer.NftTransfer[],address) (src/NFTBatchTransfer.sol#78-127) variable NFTBatchTransfer.punkContract (src/NFTBatchTransfer.sol#15) is read multiple times
Reference: https://github.com/pessimistic-io/slitherin/blob/master/docs/multiple_storage_read.md
```

- The arbitrary transferFrom() call was checked and was considered a false positive.
- All the reentrancy issues flagged by Slither were checked individually and can be considered false positives.
- No major issues found by Slither.

6.2 AUTOMATED SECURITY SCAN

Description:

Halborn used automated security scanners to assist with detection of well-known security issues and to identify low-hanging fruits on the targets for this engagement. Among the tools used was MythX, a security analysis service for Ethereum smart contracts. MythX performed a scan on the smart contracts and sent the compiled results to the analyzers to locate any vulnerabilities.

MythX results:

NFTBatchTransfer.sol

Line	SWC Title	Severity	Short Description
52	(SWC-123) Requirement Violation	Low	Requirement violation.
52	(SWC-113) DoS with Failed Call	Medium	Multiple calls are executed in the same transaction.
93	(SWC-113) DoS with Failed Call	Medium	Multiple calls are executed in the same transaction.
103	(SWC-113) DoS with Failed Call	Medium	Multiple calls are executed in the same transaction.
108	(SWC-113) DoS with Failed Call	Medium	Multiple calls are executed in the same transaction.
136	(SWC-123) Requirement Violation	Low	Requirement violation.

- MythX flagged some requirement violations, which were all false positives.
- No major issues were found by MythX.

THANK YOU FOR CHOOSING

HALBORN